

Parcial de Organización del Computador I

Ramiro Nicolás Saravia, LU: 641/19

Ejercicios:

Ejercicio 1

Sabiendo que **a1 = 0xffffffff**, ¿cuánto queda almacenado en **a2** luego de realizar la operación: **andi a2,a1,0xf00**?

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		Tipo R		
imm[11:0]						rs1		funct3		rd			opcode		Tipo I			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		Tipo S		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		Tipo B
imm[31:12]										rd			opcode			Tipo U		
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode			Tipo J		

La imagen anterior muestra los seis formatos de instrucciones básicos:

- tipo-R para operaciones entre registros
- tipo-I para immediatos cortos y loads
- tipo-S para stores
- tipo-B para branches
- tipo-U para immediatos largos
- tipo-J para saltos incondicionales

Según la tarjeta de referencia:

AND Immediate | I | ANDI rd,rs1,imm

andi a2,a1,0xf00 lo que hace es en el destino **a2** guardar el resultado de la operación AND bit a bit entre **a1** y el inmediato **0xf00**. Para hacer esto se tiene que hacer una extensión de signo a **0xf00** que quedaría a **0xfffff00**, ya que el registro destino es de 32 bits.

Luego, **a2= AND(0xffffffff, 0xfffff00) = 0xfffff00**.

Ejercicio 2

Explique de qué modo se resuelven los saltos incondicionales.

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alternativo	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

La imagen anterior muestra los 32 registros y sus nombres determinados por el ABI y la convención de preservar a través de llamadas a funciones o no. Notar que el x0 siempre tiene el valor 0, esto sirve para simplicidad en instrucciones.

Según la tarjeta de referencia:

Jump & Link	J&L	J	JAL rd, imm
Jump & Link Register		I	JALR rd, rs1, imm

Para realizar saltos incondicionales se utilizan las instrucciones **JAL** y **JALR**.

A diferencia de ORGA1 que tiene direccionamiento a palabra, RISC-V tiene direccionamiento a byte (1 byte = 8 bits). Es decir, tenemos 4 bytes en una palabra dado que una instrucción ocupa 32 bits.

El imm es lo que en la decodificación se llama offset, que sería el desplazamiento. Debido a que RISC-V tiene que una extensión de instrucciones comprimidas de 2 bytes, es decir media palabra, las instrucciones deben ser múltiplos de 2 bytes (en otras palabras, el bit menos significativo siempre es 0). Cabe mencionar que los desplazamientos, al igual que en ORGA1, son relativos (los calcula el ensamblador). Basta con poner etiquetas.

La instrucción **JAL** (jump and link) cumple 2 propósitos. En llamadas a funciones, guarda la dirección de la siguiente instrucción (PC+4) en el registro destino, que usualmente es en ra (return address). Para obtener la dirección a saltar **JAL** multiplica el offset de 20 bits por 2 (para que sea múltiplo de 2) extiende el signo y suma el resultado al PC. Si rd es omitido, se asume x1.

JAL rd, imm

$rd+ = PC + 4$; $PC = sext(imm * 2)$

Esto es lo que más se asemeja a la función CALL de ORGA1 que lo que hace es en llamadas a funciones guardar el valor del PC en el stack pointer, que acá lo guarda en una dirección de retorno y salta hacia la dirección de la etiqueta modificando el PC.

Cuando regresa de una subrutina, por ejemplo, la dirección de retorno no es útil, por lo que se utiliza el registro cero (x0) en lugar del ra como el registro de destino.

JAL x0, imm

$PC = sext(imm * 2)$

Esto es lo más se asemeja a un JMP de ORGA1 que también es un salto incondicional y que también modifica el valor del PC según la dirección de la etiqueta, solo que no realiza la multiplicación y extensión de signo.

La instrucción **JALR** (jump and link) es muy similar a **JAL**, en llamadas a funciones también guarda en rd= PC+4, si es omitido el rd también se asume que en el registro ra se guarda la dirección del retorno. La diferencia está en que en vez de llamar a una dirección específica siempre, utiliza la dirección de un registro sumado al inmediato. Esto da lugar a llamar a diferentes subrutinas en diferentes circunstancias, lo que la hace más versátil. Para hacer esto suma el valor del registro rs1 con el signo extendido del inmediato multiplicado por dos. Obviamente también hay una diferencia en la codificación dado que el formato es distinto, al agregarle un registro en el parámetro.

JALR rd, rs1, imm

$rd = PC + 4; PC = (x[rs1] + sext(imm * 2))$

Otra función que se le puede dar a esta instrucción es la de retorno (al igual que en **JAL**) de una función, usando a ra como registro origen y el registro cero como destino.

JALR x0, ra, imm

$PC = (x[ra] + sext(imm * 2))$

Ejercicio 3

¿Qué problemas puede ocasionar utilizar un registro de propósito general para el PC?

Tener un registro de propósito general para el PC implicaría que cualquier instrucción que modifique un registro puede ser, como un efecto colateral, una instrucción de branch. Además, el PC como registro complicaría la predicción de branches, lo cual es vital para un buen rendimiento del *pipeline*, dado que cualquier instrucción podría ser un branch en lugar del 10–20% de las instrucciones típicamente ejecutadas por programas. Además, implicaría un registro de propósito general menos. ARM v8, sucesor de ARM-32 dejó de usar el PC como registro de propósito general, admitiendo que fue un error.

El *pipeline* es una técnica para implementar simultaneidad a nivel de instrucciones dentro de un solo procesador. *Pipelining* intenta mantener ocupada a cada parte del procesador, dividiendo las instrucciones entrantes en una serie de pasos secuenciales, que se realizan por diferentes unidades del procesador que trabajan de forma simultánea. Aumenta el rendimiento de la CPU a una velocidad de reloj determinada. Para lograrlo, los procesadores predicen el resultado de branches con una exactitud mayor a 90%. Cuando falla la predicción, re-ejecutan instrucciones. Procesadores primitivos tenían pipelines de 5 etapas, o sea que ejecutaban 5 instrucciones traslapadas. Los más recientes tienen más de 10 etapas.

Ejercicio 4

¿Cómo se resuelve la lógica de control (branching)? ¿Qué similitudes y/o diferencias existen con la máquina Orga1?

Básicamente se utiliza lo siguiente:

Branches	Branch =	B	BEQ	rs1,rs2,imm
	Branch ≠	B	BNE	rs1,rs2,imm
	Branch <	B	BLT	rs1,rs2,imm
	Branch ≥	B	BGE	rs1,rs2,imm
	Branch < Unsigned	B	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm

BEQ: Compara dos registros y salta si son iguales

BNE: Compara dos registros y salta si son distintos

BLT: Compara dos registros y salta si el primero es menor estricto al segundo (comparación en complemento a 2)

BGE: Compara dos registros y salta si el primero es mayor o igual al segundo (comparación en complemento a 2)

BLTU: Compara dos registros y salta si el primero es menor estricto al segundo (comparación sin signo)

BGEU: Compara dos registros y salta si el primero es mayor o igual al segundo (comparación sin signo)

En los saltos condicionales vale lo mismo dicho acerca de desplazamientos relativos y la paridad de los mismos dicho en los saltos incondicionales.

Ejemplo de uso:

Si en la **memoria de intrucciones** tenemos:

0x00000000	bne x14, x17, salto
0x00000004	add x19, x20, x21
0x00000008	sub x19, x19, x03
salto:0x0000000C	...
...	...

BNE rs1, rs2, imm

Supongamos que lo que hay en el registro x14 es distinto a lo que hay en x17, entonces haría el salto hacia la etiqueta "salto". Por lo que $PC = PC + sext(imm*2)$. En el hipotético caso en el que fueran iguales $PC = PC+4$. Esta forma de realizar el salto condicional se asemeja a ORGA1, con excepción de que en vez de agregarle 4 al PC se le agregaría 1 y que en vez de comparar registros se fija en los flags para saber si realizar el salto o no.

Ejercicio 5

¿Qué son las pseudoinstrucciones? ¿Son lo mismo que microprogramación?

Además de crear código binario a partir de instrucciones, el ensamblador lo extiende para que se puedan incluir operaciones útiles para el programador de lenguaje y el escritor de compiladores. Para esto, crea funciones conocidas más cortas y de fácil entendimiento a primera vista, tales como jump, return, etc. De esta forma se simplifica mucho el set de instrucciones de RISC-V. Se dividen en dos categorías, unas que dependen del uso del registro x0 y las que no. Por ejemplo: **ret** es una pseudoinstrucción de **jalr x0, x1, 0**.

Esto representa meramente a un reemplazo ingenioso y sintáctico a instrucciones reales para el humano y no para la máquina. En OrgaSmall se utiliza microprogramación para definir el funcionamiento de una función determinada prendiendo señales a medida que avanza el clock, por lo que no tiene relación con las pseudoinstrucciones.

Ejercicio 6

¿En qué posición dentro de la instrucción se encuentran los bits de los registros destino y origen? ¿Depende del tipo de instrucción o de la instrucción en sí? ¿Por qué fue diseñado así el formato de instrucción?

31	25	24	20	19	15	14	12	11	7	6	0	
imm[31:12]								rd	0110111			U lui
imm[31:12]								rd	0010111			U auipc
imm[20:10:1 11:19:12]								rd	1101111			J jal
imm[11:0]				rs1	000			rd	1100111			I jalr
imm[12:10:5]		rs2		rs1	000			imm[4:1 11]	1100011			B beq
imm[12:10:5]		rs2		rs1	001			imm[4:1 11]	1100011			B bne
imm[12:10:5]		rs2		rs1	100			imm[4:1 11]	1100011			B blt
imm[12:10:5]		rs2		rs1	101			imm[4:1 11]	1100011			B bge
imm[12:10:5]		rs2		rs1	110			imm[4:1 11]	1100011			B bltu
imm[12:10:5]		rs2		rs1	111			imm[4:1 11]	1100011			B bgeu
imm[11:0]				rs1	000			rd	0000011			I lb
imm[11:0]				rs1	001			rd	0000011			I lh
imm[11:0]				rs1	010			rd	0000011			I lw
imm[11:0]				rs1	100			rd	0000011			I lbu
imm[11:0]				rs1	101			rd	0000011			I lhu
imm[11:5]		rs2		rs1	000			imm[4:0]	0100011			S sb
imm[11:5]		rs2		rs1	001			imm[4:0]	0100011			S sh
imm[11:5]		rs2		rs1	010			imm[4:0]	0100011			S sw
imm[11:0]				rs1	000			rd	0010011			I addi
imm[11:0]				rs1	010			rd	0010011			I slti
imm[11:0]				rs1	011			rd	0010011			I sltiu
imm[11:0]				rs1	100			rd	0010011			I xori
imm[11:0]				rs1	110			rd	0010011			I ori
imm[11:0]				rs1	111			rd	0010011			I andi
0000000		shamt		rs1	001			rd	0010011			I slli
0000000		shamt		rs1	101			rd	0010011			I srli
0100000		shamt		rs1	101			rd	0010011			I srai
0000000		rs2		rs1	000			rd	0110011			R add
0100000		rs2		rs1	000			rd	0110011			R sub
0000000		rs2		rs1	001			rd	0110011			R sll
0000000		rs2		rs1	010			rd	0110011			R slt
0000000		rs2		rs1	011			rd	0110011			R sltu
0000000		rs2		rs1	100			rd	0110011			R xor
0000000		rs2		rs1	101			rd	0110011			R srl
0100000		rs2		rs1	101			rd	0110011			R sra
0000000		rs2		rs1	110			rd	0110011			R or
0000000		rs2		rs1	111			rd	0110011			R and
0000		pred		succ	00000			000	00000		0001111	I fence
0000		0000		0000	00000			001	00000		0001111	I fence.i
000000000000					00000			000	00000		1110011	I ecall
000000000001					00000			000	00000		1110011	I ebreak
csr				rs1	001			rd	1110011			I csrrw
csr				rs1	010			rd	1110011			I csrrs
csr				rs1	011			rd	1110011			I csrrc
csr				zimm	101			rd	1110011			I csrrwi
csr				zimm	110			rd	1110011			I csrrsi
csr				zimm	111			rd	1110011			I csrrci

Figura 2.3: El mapa de opcodes de RV32I tiene la estructura de la instrucción, opcodes, tipo de formato y nombres (La Tabla 19.2 de [Waterman and Asanović 2017] es la base de esta figura).

Las posiciones del opcode, rd, rs1 y rs1 se presentan de la siguiente manera:

- Opcode, **siempre** está en los bits de 0-6
- Rd (registro destino), cuando se presenta, está **siempre** en los bits de 7-11
- Rs1 (primer registro fuente), cuando se presenta, está **siempre** en los bits 15-19
- Rs2 (segundo registro fuente), cuando se presenta, está **siempre** en los bits 20-24

La posición de los bits depende exclusivamente del tipo de instrucción, es decir, del opcode. Esta disposición ayuda a acelerar el decoding, dado que se puede a los registros antes de la decodificación (en el fetch) por su esperable posición. De esta manera se logra un mejor rendimiento.